

# springboot自动装配原理详解

1)传统ssm整合redis的时候 需要在xml的配置文件中 进行大量的配置Bean

我们在这里使用springboot来代替ssm的整合，只是通过xml的形式来整合redis

## 第一步:加入配置

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-redis</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>

<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>
```

## 第二步: 配置xml的bean的配置

```
//配置连接池
<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">
  <property name="minIdle" value="10"></property>
  <property name="maxTotal" value="20"></property>
</bean>

//配置连接工厂
<bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
  <property name="hostName" value="47.104.128.12"></property>
  <property name="password" value="123456"></property>
  <property name="database" value="0"></property>
  <property name="poolConfig" ref="poolConfig"></property>
</bean>

//配置 redisTemplate 模版类
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
  <property name="connectionFactory" ref="jedisConnectionFactory"/>
  <!--如果不配置Serializer, 那么存储的时候缺省使用String, 如果用User类型存储, 那么会提示错误User can't cast to String!>
  <property name="keySerializer">
    <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
  </property>
  <property name="valueSerializer">
    <bean class="org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer"/>
  </property>
  <property name="hashKeySerializer">
    <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
  </property>
  <property name="hashValueSerializer">
    <bean class="org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer"/>
  </property>
</bean>
```

## 第三步:导入配置

@ImportResource(locations = "classpath:beans.xml") 可以导入xml的配置文件的

```

@SpringBootApplication
@ImportResource(locations = "classpath:beans.xml")
@RestController
public class TulingOpenAutoconfigPrincipleApplication {

    @Autowired
    private RedisTemplate redisTemplate;

    public static void main(String[] args) {
        SpringApplication.run(TulingOpenAutoconfigPrincipleApplication.class, args);
    }

    @RequestMapping("/testRedis")
    public String testRedis() {
        redisTemplate.opsForValue().set("smlz","smlz");
        return "OK";
    }
}

```

2)综上所述 我们发现,若整合redis的时候通过传统的整合,进行了大量的配置.那么我们来看下通过springboot自动装配整合的对比

导入依赖:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

```

修改yml配置文件

```

spring.redis.host=47.104.128.12
spring.redis.port=6379
spring.redis.password=123456

```

直接使用(下述代码可以不要配置,为了解决保存使用jdk的序列方式才配置的)

```

@Bean
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
    RedisTemplate<Object, Object> template = new RedisTemplate<>();
    template.setDefaultSerializer(new Jackson2JsonRedisSerializer<Object>(Object.class));
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}

```

3) 传统整合和springboot自动装配 优劣势分析。 . . . . .

4) 自动装配原理前的不得不说的几个注解

4.1)通过@Import注解来导入ImportSelector组件

①: 写一个配置类在配置类上标注一个@Import的注解,

```
@Configuration
@Import(value = {TulingSelector.class})
public class TulingConfig {
}
```

②: 在@Import注解的value值 写自己需要导入的组件

在selectImports方法中 就是你需要导入组件的全类名

```
public class TulingSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        return new String[]{"com.tuling.service.TulingServiceImpl"};
    }
}
```

核心代码:

```
@RestController
public class TulingController {

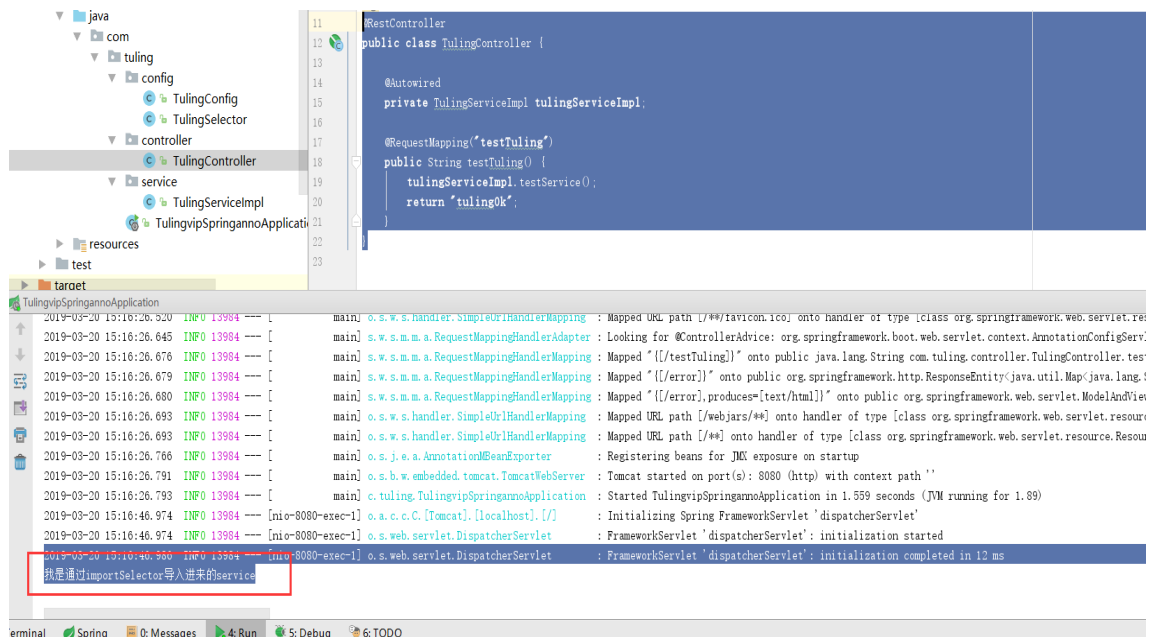
    // 自动注入 tulingServiceImpl
    @Autowired
    private TulingServiceImpl tulingServiceImpl;

    @RequestMapping("testTuling")
    public String testTuling() {
        tulingServiceImpl.testService();
        return "tulingOk";
    }
}
```

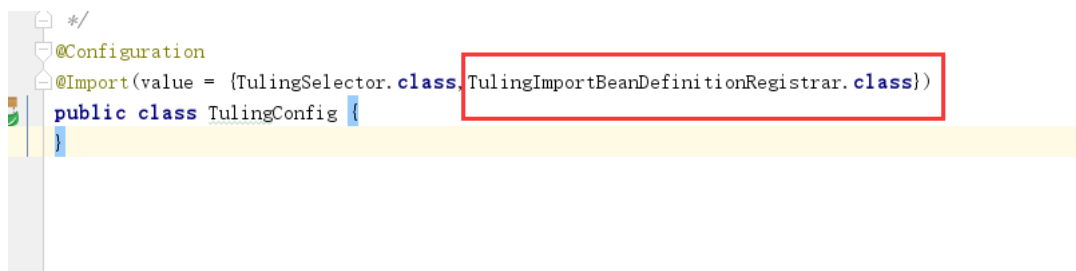
这里是没有标注其他注解提供给spring包扫描的

```
public class TulingServiceImpl {

    public void testService() {
        System.out.println("我是通过importSelector导入进来的service");
    }
}
```



## 1.2) 通过@Import导入ImportBeanDefinitionRegistrar 从而进来导入组件



核心代码:

```
public class TulingImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata annotationMetadata, BeanDefinitionRegistry beanDefinitionReg
        //定义一个BeanDefinition
        RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(TulingDao.class);
        //把自定义的bean定义导入到容器中
        beanDefinitionRegistry.registerBeanDefinition("tulingDao",rootBeanDefinition);
    }
}
```

通过ImportSelector功能导入进来的

```
public class TulingServiceImpl {
```

```
    @Autowired
    private TulingDao tulingDao;

    public void testService() {
        tulingDao.testTulingDao();
        System.out.println("我是通过importSelector导入进来的service");
    }
}
```

通过ImportBeanDefinitionRegistrar导入进来的

```
public class TulingDao {

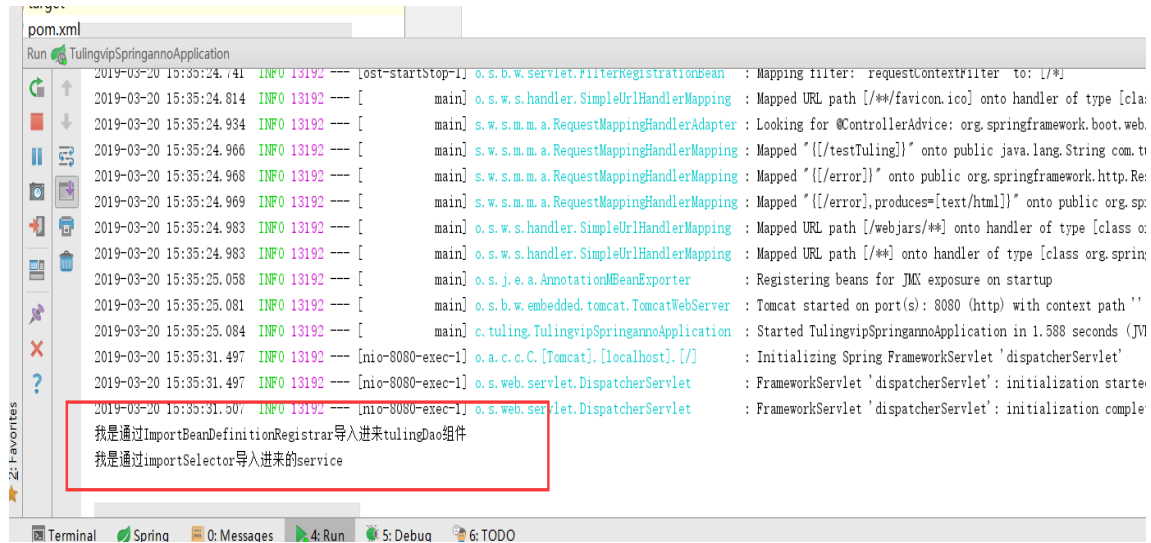
    public void testTulingDao() {
```

```

        System.out.println("我是通过ImportBeanDefinitionRegistrar导入进来tulingDao组件");
    }
}

```

测试结果:



### 1.3)spring底层条件装配的原理@Conditional

应用要求:比如我有二个组件,一个是TulingLog 一个是TulingAspect

而TulingLog 是依赖TulingAspect的 只有容器中有TulingAspect组件才会加载TulingLog

```

tulingLog组件 依赖TulingAspect组件
public class TulingLog {
}

tulingAspect组件
public class TulingAspect {
}

```

### ③:自定义条件组件条件

```

public class TulingConditional implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
        //容器中包含tulingAspect组件才返回True
        if(conditionContext.getBeanFactory().containsBean("tulingAspect")){
            return true;
        }else{
            return false;
        }
    }
}

-----该情况下会加载二个组件-----

@Bean
public TulingAspect tulingAspect() {
    System.out.println("TulingAspect组件自动装配到容器中");
    return new TulingAspect();
}

```

```

@Bean
@Conditional(value = TulingConditional.class)
public TulingLog tulingLog() {
    System.out.println("TulingLog组件自动装配到容器中");
    return new TulingLog();
}

```

-----二个组件都不会被加载-----

```

/**@Bean*/
public TulingAspect tulingAspect() {
    System.out.println("TulingAspect组件自动装配到容器中");
    return new TulingAspect();
}

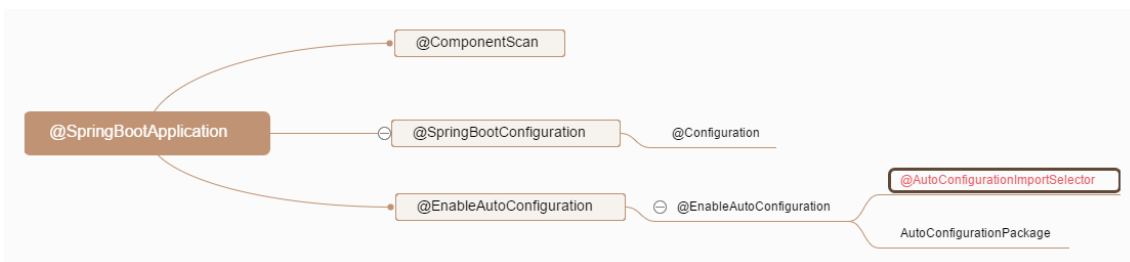
```

```

@Bean
@Conditional(value = TulingConditional.class)
public TulingLog tulingLog() {
    System.out.println("TulingLog组件自动装配到容器中");
    return new TulingLog();
}

```

## 自动装配原理分析 从@SpringBootApplication入手分析



## 那我们仔细分析

org.springframework.boot.autoconfigure.AutoConfigurationImportSelector#selectImports

```

public class AutoConfigurationImportSelector
    implements DeferredImportSelector, BeanClassLoaderAware, ResourceLoaderAware,
        BeanFactoryAware, EnvironmentAware, Ordered {

    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return NO_IMPORTS;
        }
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        //去meta-info/spring.factories文件中 查询 EnableAutoConfiguration对于值
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
        //去除重复的配置类, 若我们自己写的starter 可能存主重复的
        configurations = removeDuplicates(configurations);
    }
}

```

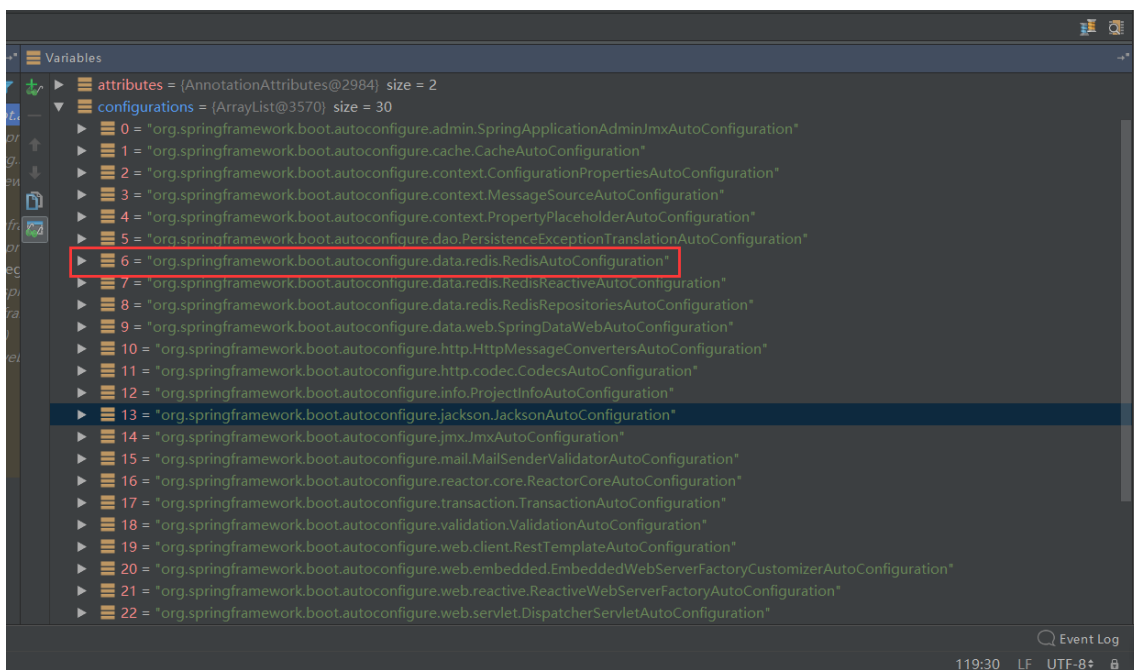
```

        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        //根据maven 导入的启动器过滤出 需要导入的配置类
        configurations = filter(configurations, autoConfigurationMetadata);
        fireAutoConfigurationImportEvents(configurations, exclusions);
        return StringUtils.toStringArray(configurations);
    }
}

//去spring.factories 中去查询EnableAutoConfrition类
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
    MultiValueMap<String, String> result = cache.get(classLoader);
    if (result != null) {
        return result;
    }

    try {
        Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                List<String> factoryClassNames = Arrays.asList(
                    StringUtils.commaDelimitedListToStringArray((String) entry.getValue()));
                result.addAll((String) entry.getKey(), factoryClassNames);
            }
        }
        cache.put(classLoader, result);
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```



然后我们分析RedisAutoConfiguration类

导入了三个组件 RedisTemplate StringRedisTemplate

JedisConnectionFactory

```
@Configuration
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionFactory.class, JedisConnectionFactory.class })
public class RedisAutoConfiguration {

    //导入redisTemplate
    @Bean
    @ConditionalOnMissingBean(name = "redisTemplate")
    public RedisTemplate<Object, Object> redisTemplate(
        RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }

    @Bean
    @ConditionalOnMissingBean
    public StringRedisTemplate stringRedisTemplate(
        RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {
        StringRedisTemplate template = new StringRedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
}

=====JedisConnectionFactory=====

@Configuration
@ConditionalOnClass({ GenericObjectPool.class, JedisConnection.class, Jedis.class })
class JedisConnectionFactory extends RedisConnectionFactory {

    private final RedisProperties properties;

    private final List<JedisClientConfigurationBuilderCustomizer> builderCustomizers;

    JedisConnectionFactory(RedisProperties properties,
        ObjectProvider<RedisSentinelConfiguration> sentinelConfiguration,
        ObjectProvider<RedisClusterConfiguration> clusterConfiguration,
        ObjectProvider<List<JedisClientConfigurationBuilderCustomizer>> builderCustomizers) {
        super(properties, sentinelConfiguration, clusterConfiguration);
        this.properties = properties;
        this.builderCustomizers = builderCustomizers
            .getIfAvailable(Collections::emptyList);
    }

    @Bean
    @ConditionalOnMissingBean(RedisConnectionFactory.class)
    public JedisConnectionFactory redisConnectionFactory() throws UnknownHostException {
        return createJedisConnectionFactory();
    }

    private JedisConnectionFactory createJedisConnectionFactory() {
        JedisClientConfiguration clientConfiguration = getJedisClientConfiguration();
```



```

        if (getSentinelConfig() != null) {
            return new JedisConnectionFactory(getSentinelConfig(), clientConfiguration);
        }
        if (getClusterConfiguration() != null) {
            return new JedisConnectionFactory(getClusterConfiguration(),
                clientConfiguration);
        }
        return new JedisConnectionFactory(getStandaloneConfig(), clientConfiguration);
    }

    private JedisClientConfiguration getJedisClientConfiguration() {
        JedisClientConfigurationBuilder builder = applyProperties(
            JedisClientConfiguration.builder());
        RedisProperties.Pool pool = this.properties.getJedis().getPool();
        if (pool != null) {
            applyPooling(pool, builder);
        }
        if (StringUtils.hasText(this.properties.getUrl())) {
            customizeConfigurationFromUrl(builder);
        }
        customize(builder);
        return builder.build();
    }

    private JedisClientConfigurationBuilder applyProperties(
        JedisClientConfigurationBuilder builder) {
        if (this.properties.isSsl()) {
            builder.useSsl();
        }
        if (this.properties.getTimeout() != null) {
            Duration timeout = this.properties.getTimeout();
            builder.readTimeout(timeout).connectTimeout(timeout);
        }
        return builder;
    }

    private void applyPooling(RedisProperties.Pool pool,
        JedisClientConfiguration.JedisClientConfigurationBuilder builder) {
        builder.usePooling().poolConfig(jedisPoolConfig(pool));
    }

    private JedisPoolConfig jedisPoolConfig(RedisProperties.Pool pool) {
        JedisPoolConfig config = new JedisPoolConfig();
        config.setMaxTotal(pool.getMaxActive());
        config.setMaxIdle(pool.getMaxIdle());
        config.setMinIdle(pool.getMinIdle());
        if (pool.getMaxWait() != null) {
            config.setMaxWaitMillis(pool.getMaxWait().toMillis());
        }
        return config;
    }
}

```

